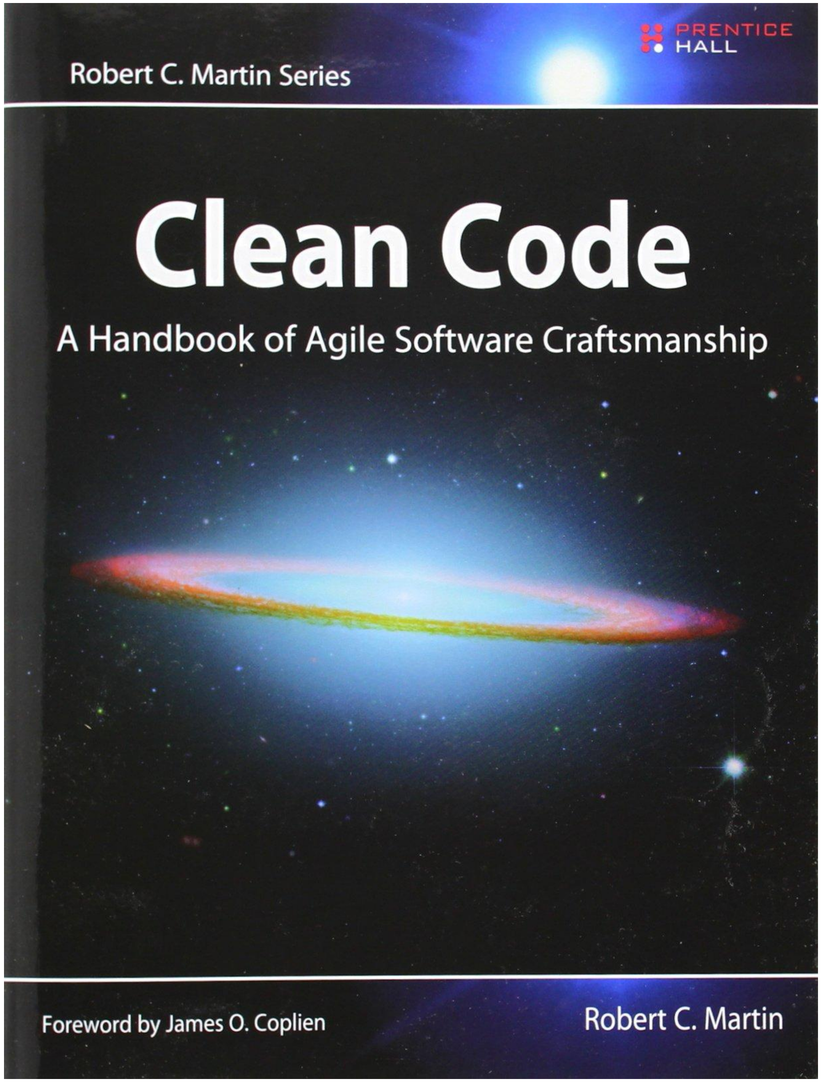
Clean Code or:
How to care for code

The Book



PRENTICE HALL
Robert C. Martin Series
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin

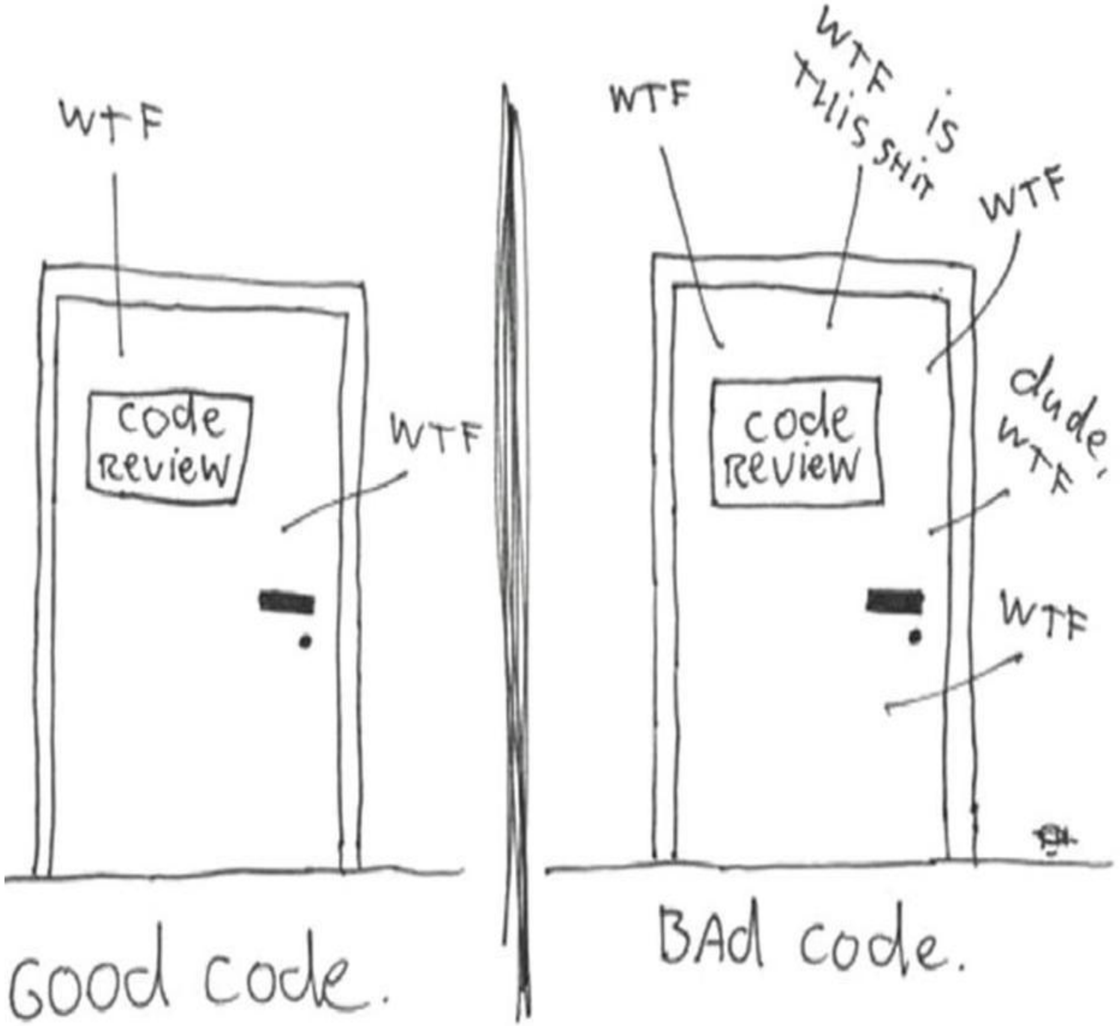
Two reasons for clean code

- You are a programmer
- You want to be a better programmer

Why code quality matters

- On average, 80% of all software work is maintenance
- On average, 90% of coding time is spent reading code

Code quality metric - WTF/s



Costs of having Bad Code

- Hard to understand and test
- Even harder to extend or maintain
- Prolongs release cycles
- Delays new features
- Ends with *The Grand Redesign in the Sky*

Excuses for Bad Code

- Short deadlines / overall workload too great
- Changing requirements
- It’s ugly but it works
- I didn’t write it, why should I fix it?
- I know it’s a mess, I’ll fix it later
  (LeBlanc’s law: *Later equals never*)

Real cause of Bad Code

Clean Code is hard work

- More than just the knowledge of principles and patterns
- Read lots of code and think hard about its good and bad sides
- Refactor mercilessly until you are satisfied with the result

How do I know Clean Code?

- Can be read, and enhanced by any coder
- Has unit and acceptance tests
- Has meaningful names
- Minimal duplication
- Provides a clear and minimal API
- Is literate

The Boy Scout Rule

- Code tends to degrade over time
- Entropy must be actively fought
- Leave the module cleaner than you found it

Names

Names

- Everywhere in software - variables, functions, arguments, classes, and packages, source files, executable files and the directories that contain them
- Since we name so much, we’d better do it well

Intention-Revealing Names

> The name of a variable, function, or class, should answer all the big questions. It should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent.
>
> ~ Robert C. Martin, *Clean Code*

Intention-Revealing Names

```
int d; // elapsed time in days

// better, but still not clear enough
int elapsedTimeInDays;

// much clearer now
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

Intention-Revealing Names

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<>();
  for (int[] x : theList)
        if (x[0] == 4) list1.add(x);
  return list1;
}

// more meaningful:
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<>();
  for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
              flaggedCells.add(cell);
  return flaggedCells;
}
```

Intention-Revealing Names

```
// even more meaningful:

public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<>();
  for (Cell cell : gameBoard)
        if (cell.isFlagged())
              flaggedCells.add(cell);
  return flaggedCells;
}
```

Use Meaningful Distinctions

```
public static void copyChars(char a1[], char a2[]) {
   for (int i = 0; i < a1.length; i++) {
      a2[i] = a1[i];
   }
  }
```

```
// easy to see what is what
public static void copyChars(char source[], char
destination[]) {
   for (int i = 0; i < source.length; i++) {
      destination[i] = source[i];
   }
}
```

Use Pronounceable Names

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = ”102”;
}

class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;;
  private final String recordId = ”102”;
}
```

Use Searchable Names

- Single letter-variables and number constants are not easily searched
- Modern IDEs allow you to find usages of a variable but number constants are harder

Use Searchable Names

```
for (int j=0; j < 34; j++) {
    s += (t[j] * 4) / 5;
}
```

```
// can be better represented as
static final int NUMBER_OF_TASKS = 34;
static final int WORK_DAYS_PER_WEEK = 5;
static final int REAL_DAYS_PER_IDEAL_DAY = 4;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] *
                       REAL_DAYS_PER_IDEAL_DAY;
  int realTaskWeeks = realTaskDays /
                       WORK_DAYS_PER_WEEK;
  sum += realTaskWeeks;
}
```

Class names

- Avoid prefixing interfaces with I
  - ShapeFactory vs. IShapeFactory
- Classes and objects should have noun or noun phrase names
  - Customer, WikiPage, Account, and AddressParser
  - Too general names like Data, Info and Processor to be used only if no better option is present

Method names

- Methods should have verb or verb phrase names
  - *postPayment*, *deletePage*, or save
  - accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the javabean standard.

Use Domain Names

- People who read your code will be programmers - use computer science terms, algorithm and pattern names freely
    - *TemplateFactory*, *MessageHandlerStrategy*, *QuickSortSorter*
- Use problem domain names to better relate the purpose of your code
    - *MessageRouter*, *AccountHolder*, *FacebookProfile*

Avoid Encodings

- Hungarian notation and other type encodings are unnecessary in modern IDEs and are only a source of code clutter
- Variable prefixes are also obsolete since modern IDEs can be configured to format the variables differently based their scope

Avoid Encodings

Hungarian notation:

```
PhoneNumber phoneString;
// name not changed when type changed!
```

Member prefixes:

```
public class Part {
    private String m_dsc; // The textual description

    void setName(String name) {
      m_dsc = name;
    }
}
```

Avoid Encodings

~~Hungarian notation:~~

```
PhoneNumber phoneNumber;
```

~~Member prefixes:~~

```
public class Part {
    private String name;

    void setName(String name) {
      this.name = name;
    }
}
```

Functions

Functions

- The first line of organization in any program
- Containers of logic

Functions - example (1)

```
public static String testableHtml(
  PageData pageData, boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
     if (includeSuiteSetup) {
        WikiPage suiteSetup =
            PageCrawlerImpl.getInheritedPage(
                  SuiteResponder.SUITE_SETUP_NAME,
                  wikiPage);
        if (suiteSetup != null) {
            WikiPagePath pagePath = suiteSetup
                  .getPageCrawler()
                  .getFullPath(suiteSetup);
            String pagePathName =
                  PathParser.render(pagePath);
```

Functions - example (2)

```
          buffer.append("!include -setup.")
                 .append(pagePathName).append("\n");
       }
   }
   WikiPage setup = PageCrawlerImpl
          .getInheritedPage("SetUp", wikiPage);
   if (setup != null) {
      WikiPagePath setupPath = wikiPage
          .getPageCrawler().getFullPath(setup);
      String setupPathName =
          PathParser.render(setupPath);
      buffer.append("!include -setup .")
          .append(setupPathName).append("\n");
   }
}
```

Functions - example (3)

```
  buffer.append(pageData.getContent());
if (pageData.hasAttribute("Test")) {
   WikiPage teardown = PageCrawlerImpl
          .getInheritedPage("TearDown", wikiPage);

   if (teardown != null) {
      WikiPagePath tearDownPath = wikiPage
             .getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser
             .render(tearDownPath);
      buffer.append("\n")
             .append("!include -teardown .")
             .append(tearDownPathName).append("\n");
   }

   if (includeSuiteSetup) {
```

Functions - example (4)

```
        WikiPage suiteTeardown =
           PageCrawlerImpl.getInheritedPage(
           SuiteResponder.SUITE_TEARDOWN_NAME,wikiPage);
        if (suiteTeardown != null) {
              WikiPagePath pagePath = suiteTeardown
                    .getPageCrawler()
                    .getFullPath(suiteTeardown);
              String pagePathName = PathParser
                    .render(pagePath);
              buffer.append("!include -teardown .")
                 .append(pagePathName).append("\n");
        }
     }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Functions - example smells

- Function is too long
- Lots of code duplication
- Name not clear enough
- Control flow too complex
    - too many nested ifs

Functions - example clean

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
      WikiPage testPage = pageData.getWikiPage();
      StringBuffer newPageContent = new StringBuffer();
      includeSetupPages(testPage, newPageContent, isSuite);
      newPageContent.append(pageData.getContent());
      includeTeardownPages(testPage, newPageContent, isSuite);
      pageData.setContent(newPageContent.toString());
  }
  return pageData.getHtml();
}
```

Functions - example cleanest

```
 public static String
renderPageWithSetupsAndTeardowns(
   PageData pageData, boolean isSuite)
 throws Exception {
   if (pageData.isTestPage())
      includeSetupAndTeardownPages(pageData,
isSuite);
   return pageData.getHtml();
 }
```

Small!

- The first rule of functions is that they should be small.
- The second rule of functions is that they should be smaller than that.

Do Only One Thing

- Functions
  - should do one thing
  - should do it well
  - should do it only

One Level Of Abstraction/f()

- Very high level of abstraction

```
test.createHtml()
```

- Intermediate level of abstraction

```
PathParser.render(pagePath)
```

- Low level

```
buffer.append(text)
```

Avoid Switch Statements

- They rarely do only one thing
- They are rarely small
- They tend to propagate throughout the code
- They usually indicate bad architecture

Avoid Switch Statements

```
class Employee...
    int payAmount() {
      switch (getType()) {
         case EmployeeType.ENGINEER:
            return _monthlySalary;
         case EmployeeType.SALESMAN:
            return _monthlySalary + _commission;
         case EmployeeType.MANAGER:
            return _monthlySalary + _bonus;
         default:
            throw new Exception("Incorrect
Employee");
      }
    }
```

Avoid Switch Statements

- Replace them with an appropriate pattern
  - AbstractFactory, Strategy, etc.
- Replace them with enums
  - Java enums can implement interfaces
- Replace them with configuration
  - maps, properties, xml, etc.

Avoid Switch Statements

```
abstract class Employe
    abstract int payAmount(Employee emp);

class Salesman
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() +
                 emp.getCommission();
    }

class Manager
  int payAmount(Employee emp) {
     return emp.getMonthlySalary() +
              emp.getBonus();
  }
```

Avoid Switch Statements

```
class EmployeeFactory {
   EmployeeType createEmployeeByType(String type) {
      switch(type) {
         case MANAGER:
            return new ManagerEmployee();
         /*...*/
      }
   }
}
```

Function arguments

- More arguments means
  - more difficult to understand
  - more difficult to test
  - often does more than one thing
  - often not simple enough
- Fix by using Parameter Object / Method Object refactorings

Function arguments

- Idealy have no arguments (niladic)
- One argument (monadic) or two (dyadic) also acceptable
- Three arguments (triadic) to be avoided where possible
- Over three (polyadic) should never be used

Niladic form

```
// Easy to test and comprehend
file.exists()
page.getHtml()
employee.calculateMonthlyPay()
```

Monadic form

```
//questions
boolean fileExits(String filePath)
// transformations
StringBuffer encodeToBase64(StringBuffer in)
// events
void passwordFailedNTimes(int times)
//setters or flags
void setVisible(boolean isVisible)
```

Dyadic form

```
writeField(name)
// is easier to understand than
writeField(outputStream, name)

// perfectly reasonable
Point p = makePoint(0, 0)
```

Triadic form

```
// bad but needed
assertEquals(message, expected, actual)
// can be replaced by fluent API
assertThat(actual).describedAs(message)
                   .isEqualTo(expected)

// possible to extract Parameter/Method Object
Circle makeCircle(double x, double y, double r);
Circle makeCircle(Point center, double r);
Circle CircleCenter#makeCircle(double r);
```

Apply Verbs To Key Words

```
write(String fieldName)

// not as clean as
writeField(String fieldName)
```

```
assertEquals(expected, actual)
// not as clean as
assertExpectedEqualsActual(expected, actual)
```

Have No Side Effects

- Misleading
- Violates the Do One Thing Rule
- Often introduces temporal coupling / function call order dependencies
  - method b must be called after method a but before method c

Avoid Output Arguments

- Arguments naturally interpreted as inputs
- Output arguments predate OOP
- In OO languages this object to be preferred over output arguments
    - make the output argument a field

DRY - Don’t Repeat Yourself

- Duplication: the root of all evil in software
- Difficult to modify / extend
  - every duplicate must be tracked down and changed, some may be overlooked
- Difficult to troubleshoot
- Goes against OO principles
  - different abstractions shouldn’t do the same thing

Classes

Small!

- The first rule of classes is that they should be small.
- The second rule of classes is that they should be smaller than that.
- The measure of size is not the number of lines but the number of responsibilities

Single Responsibility Principle

- A class (or module) should have one and only one reason to change
- Describe the class in 25 words without using "if," "and," "or," or "but."
  - if impossible, the class violates SRP
- Produces a large number of small, single-purpose classes
  - easier to test, maintain and understand

Small enough?

```
public class SuperDashboard extends JFrame {
    public Component getLastFocusedComponent(){/**/}
    public void setLastFocused(
      Component lastFocused){/**/}
    public int getMajorVersionNumber(){/**/}
    public int getMinorVersionNumber(){/**/}
    public int getBuildNumber(){/**/}

}
```

Small enough!

```
public class Version {
    public int getMajorVersionNumber(){/**/}
    public int getMinorVersionNumber(){/**/}
    public int getBuildNumber(){/**/}
}


public class FocusableDashboard extends JFrame {
    public Component getLastFocusedComponent(){/**/}
    public void setLastFocused(
      Component lastFocused){/**/}
}
```

Cohesion

- Classes should have a small number of instance variables
- Methods of a class should manipulate one or more of those variables
- The more variables a method manipulates the more cohesive that method is to its class

Cohesion

- If each field is used by each method the class is maximally cohesive
  - Rarely seen in practice
- Bad cohesion can sometimes indicate that a class should be split up into several smaller classes

Cohesion

```
public class GoodCohesionStack {
    private int topOfStack = 0;
    List<Integer> elements = new LinkedList<Integer>();

    public int size() { return topOfStack; }

    public void push(int element) {
       topOfStack++;
       elements.add(element);
    }

    public int pop() throws PoppedWhenEmpty {
       if (topOfStack == 0)
           throw new PoppedWhenEmpty();
       int element = elements.get(--topOfStack);
       elements.remove(topOfStack);
       return element;
    }
}
```

Error Handling

Exceptions, not Error Codes

- Error Codes
  - Relics of old programming languages
  - Lead to deeply nested if statements
  - Create dependency magnets
  - Require callers to check returns of every call
  - Difficult to separate happy path from error handling
  - Difficult to externalize error handlers

Exceptions, not Error Codes

```
if (deletePage(page)==E_OK)
    if (registry.deleteReference(page.name)==E_OK)
     if (configKeys.deleteKey(page.key)==E_OK)
        // do something
        else // handle error
    else // handle error
else return E_ERROR;
```

Exceptions, not Error Codes

```
try {
    deletePageAndAllReferences(page);
} catch (Exception e) {
     handleError(e);
}

private void deletePageAndAllReferences(Page page) {
     deletePage(page);
     registry.deleteReference(page.name);
     configKeys.deleteKey(page.key);
}

private void handleError(Exception e) {
    // handle error or errors
}
```

Use Unchecked Exceptions

- Checked exceptions
  - Useful only in mission-critical libraries
  - Generally do not increase robustness of software
  - Break encapsulation
  - Cause widespread boilerplate try-catch blocks
  - Cause cascading `throws` declarations throughout the call hierarchy
- Write wrapper classes around library calls and translate checked exceptions into unchecked
  -

Use Unchecked Exceptions

```
ACMEPort port = new ACMEPort(12);
try {
   port.open();
} catch (DeviceResponseException e) {
   reportPortError(e);
   logger.log(“Device response exception”, e);
} catch (ATM1212UnlockedException e) {
   reportPortError(e);
   logger.log(“Unlock exception”, e);
} catch (GMXError e) {
   reportPortError(e);
   logger.log(“Device response exception”);
} finally { /* … */}
```

Use Unchecked Exceptions

```
// Wrapper class
LocalPort port = new LocalPort(12);

try {
   port.open();
} catch (PortDeviceFailure e) {
   // Wrapped unchecked exception
   reportError(e);
   logger.log(e.getMessage(), e);
} finally { /* … */ }
```

Use Unchecked Exceptions

```
public class LocalPort {
   private ACMEPort innerPort;
   /* … */
   public void open() {
      try {
         innerPort.open();
      } catch (DeviceResponseException e) {
         throw new PortDeviceFailure(e);
      } catch (ATM1212UnlockedException e) {
         throw new PortDeviceFailure(e);
      } catch (GMXError e) {
         throw new PortDeviceFailure(e);
      }
   }
}
```

Provide Context

- Stack trace is often not enough
- Provide meaningful error messages
- If needed, also provide erroneous data
- Mention the operation that failed and the type of failure

Avoid Returning Null

- Returning Nulls
  - Forces callers to perform null-checks
  - Lowers overall code robustness
- Return empty arrays/collections/strings
- Use the Special Case pattern
  - Subclasses of the expected return type that implement the special “empty” behavior

Objects and Data Structures

Objects and data structures

- Objects
  - Hide their data behind abstractions and expose functions that operate on that data
- Data structures
  - Expose their data and have no meaningful functions
- Both have equally valid uses
  - Even in OO languages

Why variables private

- Fewer dependencies
- Easier to refactor classes and add or remove variables
- Focus is on abstractions and valid operations
- Less clutter
- Easier to enforce access rules
- Easier to provide thread-safety

Law of Demeter

- Method *m* of class C should only call methods
  - of C or of C's fields
  - of objects created by *m*
  - of objects passed as arguments to *m*
- Code that violates the Law is called a train wreck
  - `ctxt.getOptions().getScratchDir().getPath();`
- Does not apply to data structures

Comments

Comments

- Necessary evil to be used sparingly
    - More often than not, just a source of code clutter
- Don’t make up for bad code
    - Don’t comment bad code, refactor it
- Shouldn’t be used to track changes
    - Use a CVS like GitHub or Bitbucket instead
- Shouldn’t be used to hide unused code
    - Delete the code instead

Comments

- Shouldn’t be used to convey information already present in the code
- Explain Yourself in Code

```
// Is employee eligible for full benefits?
if (employee.flags & HOURLY_FLAG &&
   employee.age > 65)

if (employee.isEligibleForFullBenefits())
```

Valid uses of comments

- Legal comments
  - e.g. GNU licence declaration
- Public library/framework code documentation
  - JavaDocs API documentation
- Complex algorithm explanation
- Warnings and limitations
  - e.g. thread-safety, serialization issues
- TODO comments

Questions?

Suggested reading

- **Clean Code: A Handbook of Agile Software Craftsmanship**, Robert C. Martin, Prentice Hall, 2008.
- **The Clean Coder: A Code of Conduct for Professional Programmers**, Robert C. Martin, Prentice Hall, 2011.
- **Design Patterns: Elements of Reusable Object Oriented Software**, Gamma et al., Addison-Wesley, 1996.
- **Refactoring: Improving the Design of Existing Code**, Martin Fowler et al., Addison-Wesley, 1999.
- **The Pragmatic Programmer**, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- **Domain Driven Design**, Eric Evans, Addison-Wesley, 2003.
- **Agile Software Development: Principles, Patterns, and Practices**, Robert C. Martin, Prentice Hall, 2002.

Thank you!